Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster[1]

Gabriele Jost[*] and Haoqiang Jin
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000
{gjost,hjin}@nas.nasa.gov

Dieter an Mey
Aachen University, Center for Computing and Communication, 52074 Aachen, Germany
anmey@rz.rwth-aachen.de

Ferhat F. Hatay
Sun Microsystems High Performance and Technical Computing
Ferhat.hatay@sun.com

NAS Technical Report NAS-03-019, November 2003

Abstract

Clusters of SMP (Symmetric Multi-Processors) nodes provide support for a wide range of parallel programming paradigms. The shared address space within each node is suitable for OpenMP parallelization. Message passing can be employed within and across the nodes of a cluster. Multiple levels of parallelism can be achieved by combining message passing and OpenMP parallelization. Which programming paradigm is the best will depend on the nature of the given problem, the hardware components of the cluster, the network, and the available software. In this study we compare the performance of different implementations of the same Computational Fluid Dynamics (CFD) benchmark application, using the same numerical algorithm but employing different programming paradigms.

1. Introduction

With the advent of parallel hardware and software technologies users are faced with the challenge of choosing a programming paradigm best suited for the underlying computer architecture. With the current trend in parallel computer architectures shifting towards clusters of shared memory symmetric multi-processors (SMPs) parallel programming techniques have evolved to support parallelism beyond a single level.

Parallel programming within one SMP node can take advantage of the globally shared address space. Compilers for shared memory architectures usually support multi-threaded

---

[1] The paper was presented at the Fifth European Workshop on OpenMP (EWOMP03) in Aachen, Germany, September 2003.

[*]The author is an employee of Computer Sciences Corporation.

execution of a program. Loop level parallelism can be exploited by using compiler directives such as those defined in the OpenMP standard [6]. OpenMP provides a fork-and-join execution model in which a program begins execution as a single process or thread. This thread executes sequentially until a parallelization directive for a parallel region is found. At this time, the thread creates a team of threads and becomes the master thread of the new team. All threads execute the statements until the end of the parallel region. Work-sharing directives are provided to divide the execution of the enclosed code region among the threads. All threads need to synchronize at the end of parallel constructs. The advantage of OpenMP is that an existing code can be easily parallelized by placing OpenMP directives around time consuming loops which do not contain data dependences, leaving the source code unchanged. The disadvantage is that it is not easy for the user to optimize workflow and memory access.

On an SMP cluster the message passing programming paradigm can be employed within and across several nodes. The Message Passing Interface (MPI) [5] is a widely accepted standard for writing message passing programs. MPI provides the user with a programming model where processes communicate with other processes by calling library routines to send and receive messages. The advantage of the MPI programming model is, the user's complete control over data distribution and process synchronization, permitting the optimization of data locality and workflow. The disadvantage is that existing sequential applications require a fair amount of restructuring for parallelization based on MPI. The MPI and OpenMP programming models can be combined into a hybrid paradigm to exploit parallelism beyond a single level. The main thrust of the hybrid parallel paradigm is to combine process level coarse-grain parallelism, such as domain decomposition and fine-grain parallelism on a loop level, which is achieved by compiler directives. The hybrid approach is suitable for clusters of SMP nodes where MPI is needed for parallelism across nodes and OpenMP can be used to exploit loop level parallelism within a node.

In this study we will compare different programming paradigms for the parallelization of a selected benchmark application on a cluster of SMP nodes. We compare the timings of different implementations of the same CFD benchmark application employing the same numerical algorithm on a cluster of Sun Fire SMP nodes. The rest of the paper is structured as follows: We describe our compute platform in section 2, the different implementations of our benchmark code are described in section 3, and the performance results are presented in section 4. We then discuss related work in section 5 and conclude our study in section 6.

2. Description of the Compute Platform

For our study we used the Sun Fire cluster at the Computer Center of the University of Aachen, Germany. In this section we provide a brief system description. For more details see [11]. The system consists of :

- 16 Sun Fire 6800 nodes with 24 UltraSPARC-III Cu processors and 24GB of shared memory per node.

- 4 Sun Fire 15K nodes with 72 UltraSPARC-III Cu processors and 144GB of shared memory in each node.

The UltraSPARC-III Cu processors have a 900MHz clock rate. They are superscalar 64-bit processors with two levels of cache. The L2 off-chip cache has 8MB for data and instructions. Each CPU board contains four processors and their external L2 caches together with their local memory. The Sun Fire 6800 nodes offer a flat memory system, in other words all memory cells approximately have the same distance to each processor, with a latency of about 235ns (local access) and 274ns (remote access). The Sun Fire 15K nodes provide a cc-NUMA memory system where data locality is important. The latency for memory access within a board is about 248ns and remote access has a latency of approximately 500ns, as measured by pointer chasing. Switched Gigabit Ethernet (GE) is used to interconnect the SMP nodes. Furthermore, two tightly coupled clusters of eight Sun Fire 6800 systems are formed by interconnecting them with a proprietary high-speed Sun Fire Link (SFL) network. The Sun Fire Link is a new low-latency system area network that provides the high bandwidth needed to combine large SMP servers into a capability cluster. The network hardware exports a remote shared memory (RSM) model that supports low latency kernel bypass messaging. The Sun MPI library uses the RSM interface to implement a highly efficient memory-to-memory messaging protocol in which the library directly manages buffers and data structures in remote memory. This allows flexible allocation of buffer space to active connections, while avoiding resource contention that could otherwise increase latencies. The Sun Fire Link network achieves MPI inter-node bandwidths of almost three Gigabytes per second and MPI ping-pong latencies as low as 3.7 microseconds [10]. This compares to a latency of at least 100 microseconds and a maximum bandwidth of about 100 Megabytes per second when using GE.

For our study we used four Sun Fire 6800 nodes and one Sun Fire 15K node.

3. Benchmark Implementations

We used the Block Tridiagonal (BT) benchmark from the NAS Parallel Benchmarks (NPB) [1] for our comparative study. The BT benchmark solves three systems of equations resulting from an approximate factorization that decouples the x, y, and z dimensions of the 3-D Navier-Stokes equations. These systems are block tridiagonal consisting of 5×5 blocks. Each dimension is swept sequentially as depicted in Figure 1. We evaluated four different parallelization approaches: One based on using MPI, one based on OpenMP and two hybrid (MPI+OpenMP) strategies.

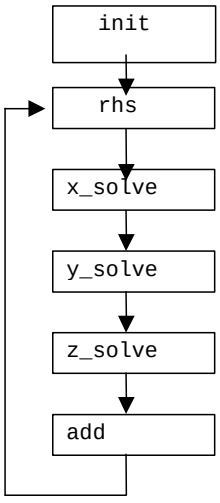

Figure 1: Structure of the BT benchmark.

The MPI implementation of BT employs a multi-partition [2] in 3-D to achieve load balance and coarse-grained communication. In this scheme, processors are mapped onto sub-blocks of points of the grid such that the sub-blocks are

evenly distributed along any direction of solution. The blocks are distributed such that for each sweep direction the processes can start working in parallel. Throughout the sweep in one direction, each processor starts working on its sub-block and sends partial solutions to the next processor before going into the next stage. An example for one sweep direction of a 2-D case is illustrated in Figure 2. The 2-D domain is divided into squares. The number within each square indicates the process number of the owner. Communications occur at the sync points as indicated by gray lines in Figure 2. We used the code as distributed in the NPB2.3, but employed several optimizations to reduce the memory requirements. The optimizations we performed are similar to those described in [3]. In the following we will refer to this implementation as BT MPI.

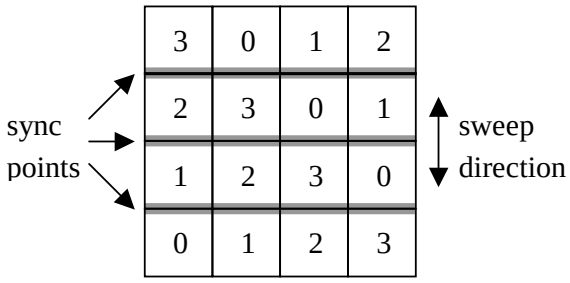

Figure 2: The multi-partition scheme in 2-D.

The OpenMP implementation is based on the version described in [3]. OpenMP directives are placed around the time consuming outermost loops. No directive nesting is employed. We will refer to this implementation as BT OMP.

We use two hybrid implementations based on different data distribution strategies. The first hybrid MPI/OpenMP implementation is based on the versions described in [3] but uses a mixed multi-dimensional parallelization strategy. The parallelization in one of the dimensions is achieved by using message-passing with a one-dimensional domain decomposition in the z-dimension. The second level of parallelism is achieved by inserting OpenMP directives on the loops in the y-dimension. Code segments for both routines are given in Figure 3. Since the data is distributed in the z-dimension, the call to `z_solve` requires communication

```
  subroutine y_solve

!$omp parallel
 do k=k_low,k_high
synchronize neighbor threads
!$omp do
   do j=1,ny
    do i=1,nx
      rhs(i,j,k)=rhs(i,j-1,k)
           + ...
    enddo
   enddo
 synchronize neighbor threads
 enddo
```

```
subroutine z_solve

!$omp parallel do
  do j=1,ny
    call receive
    do k=k_low,k_high
      do i=1,nx
      rhs(i,j,k)=rhs(i,j,k-1)
                  + ...
      enddo
    enddo
    call send
   enddo
```

Figure 3: Code segments demonstrating the tight interaction between MPI and OpenMP is routines `y_solve` (left) and `z_solve` (right) of BT Hybrid V1.

within a parallel region. The routine `y_solve` contains data dependences on the y-dimension, but can be parallelized by employing pipelined thread execution. We refer to this implementation as BT Hybrid V1.

The second hybrid implementation is based on the code as used in BT MPI, but with OpenMP directives inserted on the outermost loops in the time consuming routines. A code segment is shown in Figure 4. All communication occurs outside of the parallel regions and there is no pipelined thread execution. In each of the solver routines, there is one dimension where OpenMP directives are placed on a distributed dimension. Note that BT Hybrid V2, without enabling the OpenMP directives, is identical to the pure MPI implementation.

```
 do ib = 1, nblock
 call receive
!$omp parallel do
  do j=j_low,j_high
    do i=i_low,i_high
      do k=k_low,k_high
      lhs(i,j,k)=fac(i,j,k-1)
          + fac (i,j,k+1)...
      enddo
    enddo
   enddo
  call send
 end do
```

Figure 4: Code segment of routine `z_solve` in BT Hybrid V2.

4. Timing Results

We tested our implementations of the benchmark on three different configurations, all of them running Solaris 9, update 2:

- Four Sun Fire 6800 (SF-6800) nodes connected by a Sun Fire Link (SFL).
- Four Sun Fire 6800 (SF-6800) nodes connected by a Gigabit Ethernet (GE).
- One Sun Fire 15K (SF-15K) node.

We used the Sun ONE Studio 8 Fortran 95 compiler. In order to obtain clean and reproducible performance measurements, we had exclusive access to the Sun Fire systems. For the hybrid codes we explicitly bound all threads to separate processors with the `processor_bind` system call. We report the timings for 20 iterations of a class A benchmark corresponding to a problem size of 64x64x64 grid points.

4.1. Timing Comparison of the Benchmark Implementations

In this section we compare the scalability of each of the implementations. For the hybrid implementations we report the best timings achieved when varying the number of MPI processes and threads. The results are shown in Figures 5 and 6. The total number of available CPUs on the SF-6800 cluster is 96 and on the SF-15K node 72 CPUs are available. Some of our implementations require the number of MPI processes to be square. Therefore, we report timings for up to 81 CPUs on the SF-6800 cluster and 64 CPUs on the SF-15K node.

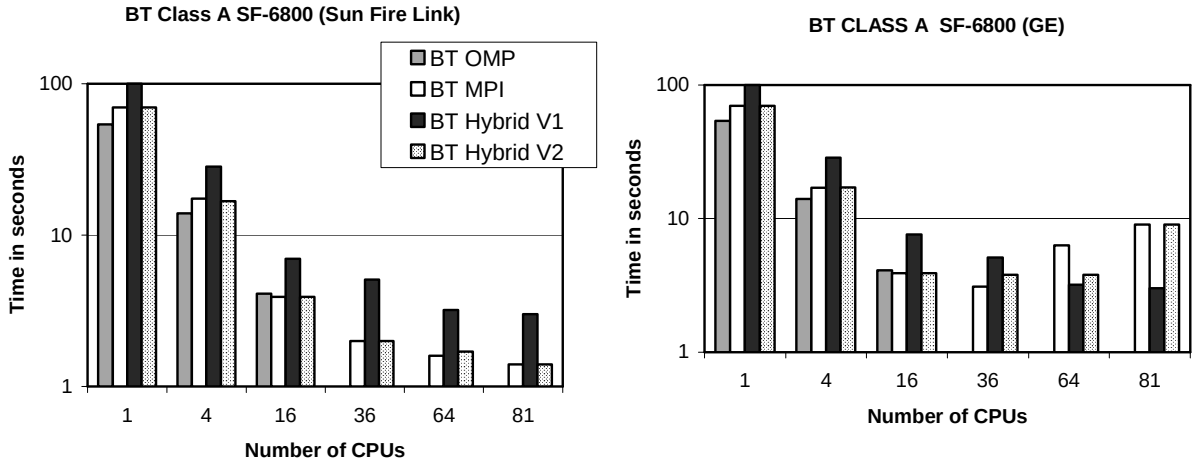

Figure 5: Timings for 20 iterations of BT Class A on a four-node SF 6800 cluster. For a small number of CPUs, the hybrid codes performed best employing only one thread per MPI process. For 64 CPUs and the slower GE network, the best timing for BT Hybrid V1 was achieved using 16 MPI processes and 4 MPI processes for BT Hybrid V2. For 81 CPUs, the best timings for both hybrid codes were achieved using 16 MPI processes with 5 threads each, employing only 80 CPUs.

The pure MPI implementation shows the best scalability for the configurations with a fast interconnect such as the Sun Fire Link (SFL) and shared memory on the SF-15K. The good performance is due to the carefully hand-optimized work distribution and synchronization resulting from the multi-partition scheme. While the OpenMP implementation also shows good scalability, it has the following disadvantages when compared to the MPI implementation:

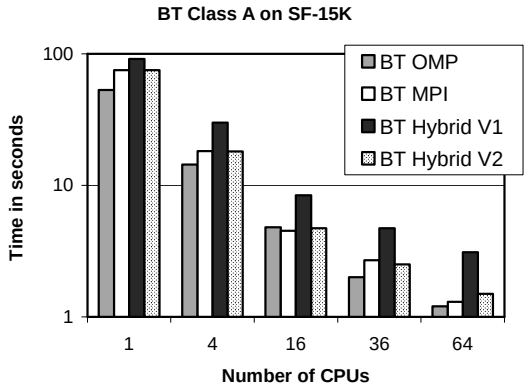

Figure 6: Timings for 20 iterations of BT Class A on the Sun Fire 15K node with 72 CPUs.

- OpenMP parallelization requires a shared address space which limits scalability to the number of CPUs within one SMP node, which is 24 for the SF-6800 and 72 for SF-15K.
- The OpenMP directives are placed only on the outer loops within a loop nest (in other words, no nested parallelism is employed). This restricts the scalability to number of inner grid points in one dimension, which is 62 for the Class A problem size.

When using the Gigabit Ethernet and a large number of CPUs)and consequently a larger number of SMP nodes), the scalability of the MPI implementation decreases. In this case, the

hybrid implementations are advantageous. We will discuss the reason for this in the following section.

4.2. Comparing Different Processes and Thread Combinations for Hybrid Codes

When using the fast Sun Fire Link network on the SF-6800 cluster or the shared memory available on the SF-15K, the hybrid codes performed best when using as many MPI processes as possible. Only in cases where the program structure limits the number of MPI processes that can be employed was is advantageous to use multiple threads per process to exploit extra parallelism. The situation is different for a slower GE network. Here it was in some cases beneficial to use a smaller number of MPI processes and increase the number of threads per process. To understand this behavior we will discuss the case of using 64 CPUs across four SF-6800 nodes. The effects of varying the number of MPI processes and OpenMP threads are shown in Figure 7. The combination of MPI processes and threads is indicated as NPxNT, where NP is the number of processes and NT is the number of threads per process.

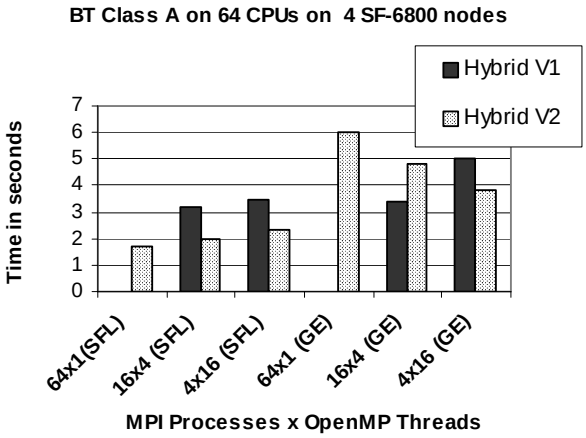

Figure 7: Effect of varying numbers of MPI processes and threads on the execution time for 20 iterations. Hybrid V2 can not be run on 64 MPI processes.

When using the Sun Fire Link interconnect, ne thread per MPI process yielded the best performance. For the GE interconnect it was advantageous to use a smaller number of MPI processes with multiple threads. Note that Hybrid V1 can not be run with 64 MPI processes, since there are only 62 grid points in each of the spatial dimensions and the implementation requires at least 2 grid points per MPI process.

As mentioned earlier, BT Hybrid V1 employs a 1-D data distribution in the z-dimension. Communication occurs in routine `z_solve` within a parallel region. Each thread exchanges partially updated data with two neighbors. The amount of data sent by each process remains the same regardless of the total number of processes used. Each process communicates data with two neighboring processes.

In BT Hybrid V2 the number of messages sent in each iteration by each process depends on the total number of processes. Increasing the number of processes increases the number of messages per process and decreases the length of the message. The number of threads per process does not influence the communication patterns, since all communication occurs outside of the parallel regions. Due to the 3-D data distribution, each process sends messages to six neighboring processes. For the problem size of class A, BT Hybrid V2 saturates the bandwidth provided by the Gigabit Ethernet (GE) when more than four MPI processes are used. If 64 MPI processes are employed, BT Hybrid V2 becomes completely communication

bound for the case of the GE network. Two parallel communications are sufficient to saturate the bandwidth of the connection, whereas it takes at least eight simultaneous transfers to saturate the bandwidth of the Sun Fire Link, as has been measured with the Pallas MPI benchmarks [4].

In BT Hybrid V1 the interaction between MPI and OpenMP is more tightly coupled than in BT Hybrid V2. MPI calls are made from within parallel regions and threads from different MPI processes have to synchronize with each other. This greatly increases overhead introduced by OpenMP such as barrier synchronization. We have also noticed lock contention when multiple threads make calls to the MPI library, indicating that these calls get serialized in order to make them thread-safe. This leads to the conclusion that increasing the number of threads in BT Hybrid V1 not only increases the OpenMP overhead, but also increases the time spent in MPI calls. These problems do not occur in BT Hybrid V2. While the use of OpenMP directives introduces the usual overhead involved with the forking and joining of threads and thread barrier synchronization at the end of parallel regions, there is no negative effect on the MPI parallelization. In Table 1 we have summarized the MPI and OpenMP characteristics for the runs from Figure 7.

	Total #bytes	#sends	avg. msg. length	OpenMP	MPI
BT V1 4x16	157,728,000	14,880	10,600 bytes	23%	25%
BT V1 16x4	630,912,000	59,520	10,600 bytes	17%	4%
BT V2 4x16	111,705,600	960	116,360 bytes	15%	7%
BT V2 16x4	352,435,200	7,680	45,890 bytes	5%	19%
BT V2 64x1	814,080,000	61,440	13,253 bytes	0%	125%

Table 1: Summary of MPI and OpenMP characteristics for the hybrid codes. The last two columns indicate the percentage of time spent in MPI calls and OpenMP barrier synchronization versus the user CPU time.

We conclude that Hybrid V1 can employ more MPI processes than Hybrid V2 before saturating the GE network. The overhead introduced by increasing the number of threads per process is greater for Hybrid V1 than for V2, which is due to the tight interaction between MPI and OpenMP.

4.3. Usage of Multiple SMP Nodes

In Figure 8 we show timings for BT Hybrid V2 for 16 CPUs when running on 1, 2, and 4 SMP nodes. We compare runs using only one thread per MPI process (16x1) and runs employing four processes with four threads each. The MPI processes are distributed evenly between the SMP nodes. For the relatively small number of CPUs it was always best to use only one thread per MPI process for the Fire Link as well as the GE interconnect. We observe

that using the Sun Fire Link (SFL) is as fast as using the shared memory within one of the SMP nodes.

5. Related Work

There are many published reports on the comparison of different programming paradigms. We can only name a few of them. A comparison of message passing versus shared memory access is given in [8] and [9]. The studies focus on shared memory architectures. Some aspects of hybrid programming on SMP clusters are discussed in [7]. An evaluation of MPI using the Sun Fire Link network is given in [10].

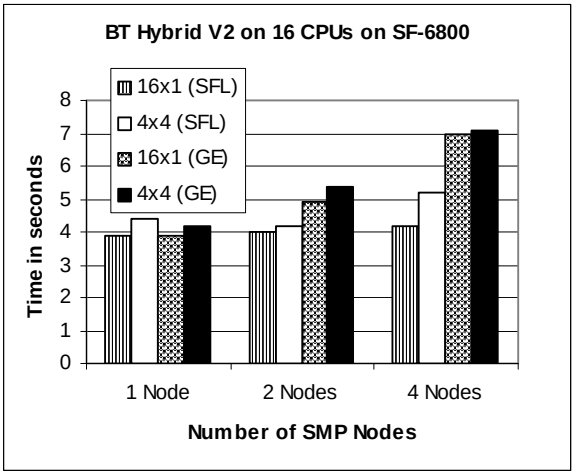

Figure 8: Timings for 20 iterations of BT Hybrid V2 on 16 SF-6800 CPUs.

6. Conclusions and Future Work

We have run several implementations of the same CFD benchmark code employing different parallelization paradigms on a cluster of SMP nodes. When using the high-speed interconnect or shared memory, the pure MPI paradigm turned out to be the most efficient. A slow network lead to a decrease in the performance of the pure MPI implementation. The hybrid implementations showed different sensitivity to network speed, depending on the parallelization strategy employed. The benefit of the hybrid implementations was visible on a slow network.

The hybrid parallelization approach is suitable for large applications with an inherent multilevel structure, such as multi-zone codes. For codes like the BT benchmark, where parallelization occurs only on one or more of the spatial dimensions, the use of either process level parallelization or OpenMP is, in general, more appropriate. We plan to conduct a study using multi-zone versions of the NAS Parallel Benchmarks [12], which are more suitable for the exploitation of multilevel parallelism.

Acknowledgements

This work was partially supported by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation. We would like to thank the Center for Computing and Communication of the University of Aachen for their support of our study. We thank Eugene Loh and Larry Meadows from Sun Microsystems for their help with using the Sun performance analysis tools.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995. http://www.nas.nasa.gov/Software/NPB/.

[3] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance", *NAS Technical Report NAS-99-011*, 1999

[4] D. an Mey, "The Sun Fire Link – First Experiences", Sun HPC Consortium, Heidelberg, 2003, http://www.hpcconsortium.org/.

[5] MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich/.

[6] OpenMP Fortran Application Program Interface, http://www.openmp.org/.

[7] Rabenseifner, R., "Hybrid Parallel Programming: Performance Problems and Chances", Proceedings of the 45th Cray User Group Conference, Ohio, May 12-16, 2003.

[8] H. Shan, J. Pal Singh, "A comparison of MPI, SHMEM, and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessor", *International Journal of Parallel Programming*, Vol. 29, No. 3, 2001.

[9] H. Shan, J. Pal Singh, "Comparison of Three Programming Models for Adaptive Applications on the Origin 2000", *Journal of Parallel and Distributed Computing 62, 241-266, 2002.*

[10] S. J. Sistare, C. J. Jackson, "Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect", *Proceedings of Supercomputing 2002.*

[11] The Sun Fire SMP-Cluster, http://www.rz.rwth-aachen.de/computing/info/sun/primer.

[12] R. Van der Wijngaart and H Jin, "NAS Parallel Benchmarks, Multi-Zone Versions", *NAS Technical Report NAS-03-010*, NASA Ames Research Center, Moffett Field, CA, 2003.